Project 3: Correlation Attack

Group 06: Fredrick Nilsson

December 6, 2023

Exercise 1

I found the following initial states:
K_1: [1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1] with a correlation 0.69948184
K_2: [0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1] with a correlation 0.7098446
K_3: [1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0] with a correlation 0.78238344
For the given output sequence: 10010000111011001011010110010010110101 10101110011010101010010000001100100110001110111011011100010001111111010101000001000010101110110000111100011100001010000110010011000001000110001110111101010

Exercise 2

With a correlation attack, we have to search through 2^L for each LFSR of length L. For the LFSRs in this example it adds up to $2^{13} + 2^{15} + 2^{17}$ states to search though, which we assumes takes T time.
With exhaustive key search we have to search through all states of each of the LFSRs for each state in the others which equates to $2^{13} \times 2^{15} \times 2^{17}$.
We can therefore calculate the amount of time it takes as $\frac{2^{13} \times 2^{15} \times 2^{17}}{2^{13} + 2^{15} + 2^{17}} \times T = \frac{4294967296}{21} \times T \approx 2.05 \times 10^8 \times T$.
For example, on my Macbook Air (2021), it takes approximately 912 milliseconds to run the correlation attack. So a rough estimate of the time it would take to run exhaustive key search is $2.05 \times 10^8 \times 0.912 \approx 1.87 \times 10^8$ seconds, which is about 5.9 years.

Source code

```
use rayon::prelude::*;

pub fn exercise1() {
    let num = include_str!("../task06.txt").chars().map(|c|
    c.to_digit(10).unwrap() as u8).collect::<Vec<u8>>();

    // Primitive polynomials
```

```
    let prim: &Vec<Vec<u8>> = &vec![
        vec![1,0,1,1,0,0,1,1,0,1,0,1,1],
        vec![1,0,1,0,1,1,0,0,1,1,0,1,0,1,0],
        vec![1,1,0,0,1,0,0,1,0,1,0,0,1,1,0,1,0]];

    // Generate de Bruijn sequences for the primitive polynomials
    let seq: &Vec<Vec<u8>> = &prim.into_par_iter()
        .map(|p| lfsr(&p, vec![0; p.len()], 2_usize.pow(p.len() as u32))).collect();

    // Find the position of the states with maximum correlation to the given number
    let pos = seq.into_par_iter().map(|s| max_p(&s, &num)).collect::<Vec<usize>>();

    // Get the specific states of the sequence with the maximum correlation
    let states = seq.iter().zip(pos.iter()).map(|(s, i)| s[*i..*i+num.len()].to_vec()).collect::<Vec<Vec<u8>>>();

    // Confirm that the three sequences generate the given number
    if check_seqs(num.clone(), &states){
        println!("Found!");
        println!("State 1: {:?}", seq[0][pos[0]..pos[0]+prim[0].len()].to_vec());
        println!("State 2: {:?}", seq[1][pos[1]..pos[1]+prim[1].len()].to_vec());
        println!("State 3: {:?}", seq[2][pos[2]..pos[2]+prim[2].len()].to_vec());
    } else {
        println!("Not found!");
    }
}

// Checks if the sequences generate the given number with majority vote
```

```
fn check_seqs(num: Vec<u8>, seq: &Vec<Vec<u8>>) -> bool{
    num == seq[0].iter()
        .zip(seq[1].iter())
        .zip(seq[2].iter())
        .map(|((x,y),z)| (*x+*y+*z) / 2)
        .collect::<Vec<u8>>()
}

// Finds the position of the maximum correlation between the given sequence and the given number
fn max_p(seq: &Vec<u8>, num: &Vec<u8>) -> usize {
    let mut dists: Vec<f32> = Vec::new();
    for i in 0..(seq.len()-num.len()) {
        let j = i + num.len();
        let state = seq[i..j].to_vec();
        let dist = distance(&state, &num);
        dists.push(dist);
    }
    let (pos, _max) = dists.iter().enumerate().max_by(|(_, x),(_, y)| (0.5-**x).abs().partial_cmp(&(0.5-**y).abs()).unwrap()).unwrap();
    println!("Max: {}", _max);
    pos
}


// Generates a lfsr sequence of length len, starting with the state init and using prim as the primitive polynomial
fn lfsr(prim: &Vec<u8>, mut seq: Vec<u8>, len: usize) -> Vec<u8> {
    for _ in 0..len {
        let last = seq.as_slice()[seq.len()-prim.len()..].to_vec();
        if last[1..].to_vec() == vec![0_u8; prim.len()-1] {
            // special case of 0 state
            seq.push(if last[0]==1 {0} else {1});
        } else {
            // general case
```

```
            seq.push(and(last,prim.clone()).iter().sum::<u8>() % 2);
        }
    }
    seq
}

// Bitwise and of two vectors
fn and(a: Vec<u8>, b: Vec<u8>) -> Vec<u8> {
    a.iter().zip(b.iter()).map(|(a,b)| a&b).collect::<Vec<u8>>()
}

// Calculates the distance between two vectors
fn distance(a: &Vec<u8>, b: &Vec<u8>) -> f32 {
    1.0 - hamming(a,b) as f32 / a.len() as f32
}

// Calculates the hamming distance between two vectors
fn hamming(a: &Vec<u8>, b: &Vec<u8>) -> u8 {
    a.iter().zip(b.iter()).map(|(a,b)| a^b).sum()
}
```